Chapter 6

Constant Propagation

Mark N. Wegman
Kenneth Zadeck **December 28, 1992**

6.1 Introduction

Constant propagation is a well-known global flow analysis and optimization problem. The goal of constant propagation is to discover values that are constant in all possible executions of a program and to propagate these constant values as far forward through the program as possible. Expressions whose operands are all constants can be evaluated at compile time and the results propagated further.

While the constant-propagation problem is easily shown to be undecidable in general (see [KU77], for example), there are many reasonable instances of the problem that are decidable and for which computationally efficient algorithms exist. We present two such algorithms in this chapter. Both algorithms are *conservative* in the sense that not all constants may be found, but each one found is constant over all possible executions of

the program. After some preliminaries, the algorithms are presented in Sec. 6.3. In Sec. 6.4, a common implementation problem is discussed.

Constant-propagation techniques serve several purposes in optimizing compilers:

- Expressions evaluated at compile time need not be evaluated at execution time. If such expressions are inside loops, a single evaluation at compile time can save many evaluations at execution time.
- Code that is never executed can be deleted. Unreachable code (a form of dead code) is discovered by identifying conditional branches that always take the same branch path.
- Detection of paths never taken simplifies the control flow of a program. The simplified control structure can aid the transformation of the program into a form suitable for vector processing [FP82], [Pra78] or parallel processing [Ell86].
- Constant propagation can be done over a variety of domains, for example, over the types of variables [JM76].

6.2 Preliminaries

This section introduces the mathematical notation used to represent programs and values.

6.2.1 Graph Definitions

Since an algorithm's performance is usually specified in terms of the size of its input, it is necessary to define some common measures of program size:

N is the number of assignment statements plus the number of expressions whose values are branched on in the program. For notational convenience, the basic blocks are minimal; each node in the control-flow graph contains one expression. This number also closely approximates the number of definition sites in the program, since most statements assign a value.

E is the number of edges in the control-flow graph. A usually reasonable approximation for E is twice N, since branch statements typically have only two successors, though E may be as large as N^2.

V is the number of variables in the program.

A program consists of three types of nodes: *branch nodes, assignment nodes,* and a unique *entry node.*[1] Branch nodes are potential deviations in control flow through a program. An expression is evaluated at a branch node and control is subsequently transferred to another node; for simplicity, assume that such expressions have no side effects. Assignment nodes are sites at which variables are defined in terms of other variables and constants; for simplicity, assume that only scalar (i.e., non-subscripted) variables participate in assignment nodes. In procedures with multiple entry points, the entry node has out-edges to all procedure entry nodes.

6.2.2 The Lattice for Constant Propagation

The output of a constant-propagation algorithm is an *output assignment* of lattice elements to variables at each node in the program. Let all variables defined or used within a given assignment or branch node be characterized by a *lattice element* that represents compile-time knowledge about the value of such variables during execution of the algorithm. As depicted in Fig. 6.1, the lattice element can be any of three types: the highest element is *top*, $\top$, the lowest is *bottom*, $\bot$, and all elements in the middle are *constant*, $\mathcal{C}$. There may be an infinite number of $\mathcal{C}$ lattice elements, each corresponding to a different constant. In the lattice-theoretic sense, no constant is higher or lower than any other. Each node of the program has cells, called *LatticeCells*, to hold the lattice elements; the values stored in these elements change as the algorithm progresses. The LatticeCells are associated with the results and operands of the expressions; the details of the association depend on the particular algorithm.

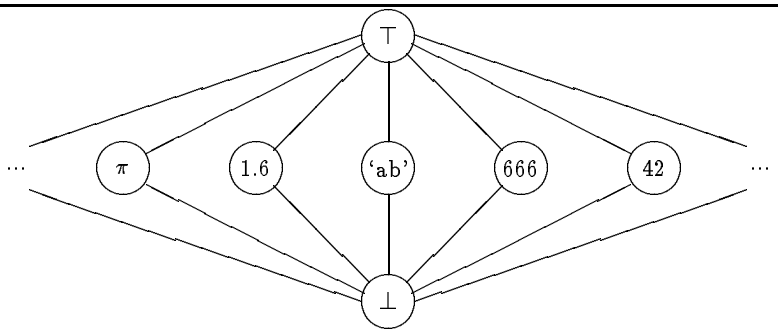

Figure 6.1. The three-level constant propagation lattice.

At the end of constant propagation, a constant $\mathcal{C}$ assigned to a Lattice-

[1] The exit node defined in Chapter 3 is irrelevant to this algorithm.

Cell means that for all possible executions of the program, the associated result or operand always has the value $\mathcal{C}$ after that node has executed. Assigning $\bot$ means that a constant value cannot be guaranteed, and assigning $\top$ means that the variable may be some (as yet) undetermined constant. Upon termination of a constant propagation algorithm, all LatticeCells are either $\mathcal{C}$ or $\bot$ at executable nodes.

The algorithms start with the optimistic assignment of $\top$ to the LatticeCells of all operands of expressions at all nodes except the entry node. If the variable has an explicit initializer or if the language specifies an implicit initializer (e.g., in Lisp, all cells are initialized to `NIL`), then that value is used at the entry node. The algorithms may obtain better information if $\top$, rather than $\bot$, is assigned to the values of uninitialized variables at the entry of the program. In languages in which the use of an uninitialized variable is allowed but undefined (as in Fortran), the use of $\top$ may make the analysis yield an incorrect result; thus, each uninitialized variable must be assigned $\bot$. On the other hand, if the language forbids such uses, the uninitialized variables may be assigned $\top$.

The algorithms proceed by lowering (in the lattice-theoretic sense) the LatticeCells of the operands and results at each node as more information is discovered, a process that continues until a fixed point has been achieved. The additional information is inserted by applying the meet ($\sqcap$) rules shown in Fig. 6.2, where each of the operand values at a node corresponds to the conditions prior to the execution of the statement. These rules ensure that the value at the join point is no higher than the value entering from any of the predecessors.

$$
\begin{array}{lcll}
\texttt{any} & \sqcap & \top & = \quad \texttt{any} \\
\texttt{any} & \sqcap & \bot & = \quad \bot \\
\mathcal{C}_i & \sqcap & \mathcal{C}_i & = \quad \mathcal{C}_i \\
\mathcal{C}_i & \sqcap & \mathcal{C}_j & = \quad \bot \quad \text{if } i \neq j
\end{array}
$$

Figure 6.2. Rules for $\sqcap$.

If a variable is not the target of an assignment statement in a node, then its value is unchanged by the node. If the node is an assignment statement, then the value of the result and the values of operands of other nodes may change and are reevaluated according to the *expression evaluation rules*: the values of the operands of an expression correspond to the values of the variables at the entrance to the node, and the result is determined from the values of variables that change during the execution of the node.

Usually, if the node is an assignment and any of the variables used in

its expression portion has a value of $\perp$, the value exiting the assignment statement for that variable is $\perp$. If all values used in its expression portion are constant, the value of the assigned variable is the value of the expression when evaluated with those constant values. Otherwise the value assigned is $\top$.

For certain operators, however, we can give special expression rules that yield better information. For example, if the operator is a logical "or" or $\vee$ and one of the operands is known to be `TRUE`, then the value of the expression is `TRUE` whether or not the other operand is $\perp$. These rules are given in Fig. 6.3.

```
any  ∨  TRUE  =  TRUE
any  ∧  FALSE =  FALSE
```

Figure 6.3. Special rules for $\wedge$ and $\vee$.

Information can sometimes be derived from the equality tests that control conditional branches [AC72]. On entrance to the `then` branch in Fig. 6.4, the value of `I` is `5`. We can modify the program so that we can derive this information by inserting extra nodes containing assignments between the conditional and the entrance to the `then` branch. The variable `I` is assigned the `join` of the LatticeCells of `I` and `J` (a similar assignment to `J` is also added). These are not assignment statements in the ordinary sense, because they are not used to cause changes in the program's state and need not be executed; rather, they are used to model changes in the assertions about the program state, and thus are in the intermediate code.

```
J ← 5
if I = J then I ← I + 1
```

Figure 6.4. A conditional branch where information about `I` can be derived.

6.3 Constant Propagation Algorithms

In this section we present two algorithms for determining constants. The first algorithm, *Conditional Constant* (CC), is a variant of Wegbreit's Algorithm 3.1 [Weg75] and is presented in Sec. 6.3.1. CC discovers all constants that can be found by evaluating all conditional branches with all

constant operands. The technique is an enhanced version of the algorithm in Fig. 4.11. The attraction of CC is that it propagates the values in such a way that when conditional branches are found to have a constant conditional expression, the search for constants can ignore parts of the program that are never executed. The algorithm does unreachable-code elimination in combination with constant propagation. CC has two advantages over the technique in Fig. 4.11: first, CC may run faster than Fig. 4.11 since it need not evaluate the sections of the program that are never executed; second, values created in the unreachable areas cannot possibly kill potential constants, and thus CC can find more constants than the other algorithm can.

The second algorithm, *Sparse Conditional Constant* (SparseCC), was developed by Wegman and Zadeck [WZ85, WZ91] and is presented in Sec. 6.3.2. SparseCC finds the same class of constants as CC, but is much faster.

6.3.1 Conditional Constant

Wegbreit's Algorithm 3.1 [Weg75] is a general algorithm for performing global flow analysis that takes conditional branches into account. In this section, we specialize Wegbreit's general algorithm to perform constant propagation and call the result *Conditional Constant* (CC). CC finds all values that can be proven to be constant subject to one constraint: only one value for each variable is maintained along each path in the program. CC uses the control-flow graph for propagation of values. At each node in the program, two LatticeCells are associated with the value of every variable in the program, one with the value at entry to the node and the other with the exit. The process of visiting a node involves examination of every LatticeCell at that node. Initially the entry node is placed on the worklist. A node is chosen from the worklist, removed from the worklist, and examined. The lattice value for variable `V` stored at the entry of the examined node becomes the meet of values for `V` at the exits of preceding nodes. The statement is evaluated on the basis of the new entering values.

This may cause the value of a variable assigned in the node to differ from the value associated with the variable at the exit LatticeCell. In that case, all nodes following the examined node must be examined, so they are added to the worklist. If no exit LatticeCells change, then no nodes are added to the worklist. The process repeats until the worklist is empty.

Consider the example in Fig. 6.5. The simple iteration of Sec. 4.11 is not capable of discovering that `J = 1` since it makes no assumptions about the possible branch directions. Since `I` is always `1`, however, the condition always takes the `TRUE` branch and `J` is always equal to `1`. Such code may be the result of procedure integration or abstract data type compilation.

```
I ← 1
...
if I = 1
   then J ← 1
   else J ← 2
```

Figure 6.5. A conditional constant definition.

To exploit this knowledge about conditional branches, we do not propagate values along all control flow graph edges. Rather, CC defers the evaluation of any control flow graph edge until it is marked as executable. Each control flow graph edge is initially marked as not executable. Control flow graph edges are marked executable by symbolically executing the program, beginning with the entry node. Whenever an assignment node is executed, the out-edge in the control flow graph leaving that node is marked as executable and added to the worklist. Whenever a branch node is executed, the expression controlling the conditional is evaluated and we determine which branch(es) may be taken. If the expression evaluates to $\perp$, then all branches may be taken, and the edges corresponding to these branches are added to the worklist. If the expression evaluates to $\mathcal{C}$, only one branch can be taken, and the associated edge is added to the worklist.

This algorithm is able to ignore any definition that reaches a use via a control flow graph edge that is never executed. Thus, this algorithm accomplishes a form of *dead-code elimination* called *unreachable-code elimination*.[2]

Since the lattice value of each variable can only be lowered twice, each node may be visited at most $2 \times V \times I$ times, where I is the number of in-edges into that node. Thus, the time required for CC is $O(E \times V)$ node visits and V operations during each node visit. This results in a worst-case running time of $O(E \times V^2)$. The best-case running time may be $O(E \times V)$. It is our experience that the worst case rarely occurs. The space required is $O(N \times V)$. This algorithm is expected to have better average-case complexity, however, since it can ignore parts of the program that will never be executed.

Many optimizing compilers repeatedly execute constant propagation and unreachable-code elimination since each provides information that improves the other. CC solves this problem in an elegant way by combining

[2] Two classical techniques are called dead-code elimination. The goal of the first, *unreachable-code elimination*, is to eliminate code that can never be executed. The goal of the other, *unused-code elimination*, is to delete sections of code whose results are never used (see Sec. 15.2). Each of these techniques finds a different class of dead code, and neither subsumes the other.

the two optimizations. Additionally, the algorithm gets better results than are possible by repeated applications of the separate algorithms.

6.3.2 Sparse Conditional Constant

When CC visits a node, it applies a function that takes as input the values of all variables at the entrance of the node and produces the set of values for all variables at the exit of the node. In this section we reformulate that notion in order to produce an algorithm that gets the same result as CC but much faster. *Sparse Conditional Constant* (SparseCC) finds the same constants as CC. It achieves a speedup over CC by using a sparse representation, the SSA graph, to propagate the values through the program.

Once a program is in SSA form, we add connections called *SSA edges*. Each connection goes from the unique point where a variable is given a value to a use of that variable. SSA edges are essentially def-use chains (see Sec. 4.6.1) in the SSA program.

When the SSA graph is constructed, ϕ-functions are inserted at some join nodes. The meaning of a ϕ-function is that if control reaches the node in the control flow graph along its ith in-edge, the result of the ϕ-function is the value of its ith operand. In this algorithm, the meet operator is applied only to those operands of the ϕ-function that correspond to the control flow graph edges marked as executable. Those that are not executable effectively have the value $\top$.

This algorithm uses two worklists: *FlowWorkList* is a worklist of control flow graph edges and *SSAWorkList* is a worklist of SSA edges.

SparseCC works as follows:

1. Initialize the FlowWorkList to contain the edges exiting the entry node of the program. The SSAWorkList is initially empty.

   Each control flow graph edge has an associated flag, the *Executable-Flag*, that controls the evaluation of ϕ-functions in the destination node of that edge. This flag is initially `FALSE` for all edges.

   Each LatticeCell is initially $\top$.

2. Halt execution when both worklists become empty. Execution may proceed by processing items from either worklist.

3. If the item is a control flow graph edge from the FlowWorkList, then examine the ExecutableFlag of that edge. If the ExecutableFlag is `TRUE` do nothing; otherwise:

   (a) Mark the ExecutableFlag of the edge as `TRUE`.

   (b) Perform Visit-ϕ for all of the ϕ-functions at the destination node.

(c) If only one of the ExecutableFlags associated with the incoming control flow graph edges is `TRUE` (i.e., if this is the first time this node has been evaluated), then perform VisitExpression for the expression in this node.

(d) If the node only contains one outgoing control flow graph edge, add that edge to the FlowWorkList.

4. If the item is an SSA edge from the SSAWorkList and the destination of that edge is a ϕ-function, perform Visit-ϕ.

5. If the item is an SSA edge from the SSAWorkList and the destination of that edge is an expression, then examine ExecutableFlags for the control flow edges reaching that node. If any of them are `TRUE`, perform VisitExpression. Otherwise, do nothing.

The value of the LatticeCell associated with the output of a ϕ-function is defined to be the meet of all arguments whose corresponding in-edges have been marked executable. It is computed by Visit-ϕ. Visit-ϕ is called whenever the value of the LatticeCell associated with one of its operands is lowered or when the ExecutableFlag associated with one of the in-edges becomes `TRUE`.

Visit-ϕ is defined as follows: The LatticeCells for each operand of the ϕ-function are defined on the basis of the ExecutableFlag for the corresponding control flow edge, which is defined as follows:

`executable` The LatticeCell has the same value as the LatticeCell at the definition end of the SSA edge.

`not-executable` The LatticeCell has the value $\top$.

VisitExpression is defined as follows: Evaluate the expression obtaining the values of the operands from the LatticeCells where they are defined and using the expression rules defined in Sec. 6.2.2. If this changes the value of the LatticeCell of the output of the expression, do the following:

1. If the expression is part of an assignment node, add to the SSAWorkList all SSA edges starting at the definition for that node.

2. If the expression controls a conditional branch, some outgoing flow graph edges must be added to the FlowWorkList. If the LatticeCell has value $\bot$, all exit edges must be added to the FlowWorkList. If the value is $\mathcal{C}$, only the flow graph edge executed as the result of the branch is added to the FlowWorkList.[3]

[3] The value cannot be $\top$, since the earlier step in VisitExpression will have lowered it.

Asymptotic Complexity

The time complexity of this algorithm is proportional to the size of the SSA graph. The SparseCC algorithm requires that each SSA edge be examined at least once and at most twice. The examinations occur when the value of the edge's definition site is lowered to either $\mathcal{C}$ or $\perp$.

6.4 Def-Use Chains

Many constant propagation algorithms work on a graph of *def-use chains*. Def-use chains can cause two problems with the algorithms presented here. A def-use chain is a connection from a *definition site* for a variable to a *use site* for that variable. A definition site for a variable is a statement that assigns to the variable. A use site is normally an operand of an expression. There is a def-use chain between a definition site and a use site if the use site can be reached from a definition site along the control flow graph without passing through another definition site for that variable.

If SparseCC is performed using def-use chains rather than the SSA graph, some constants will be missed, because def-use chains exist along paths that are not executable. Consider the program shown in Fig. 6.6. Statement (b) provides the only possible value for statement (c), because the path through (b) is the only path to (c) (we know this because the condition must always be `TRUE`). A def-use-chain version of SparseCC will not find this because it applies the def-use chain that starts from (a) and does not know that the value is really killed by (b).

```
(a)  I ← 1
     J ← 2
     if J = 2
(b)     then I ← 3

(c)  ... ← I
```

Figure 6.6. Constant not found with def-use chains.

Another shortcoming of def-use chains is related to the size of the graph. In def-use chains, many definitions can reach a use. The number of def-use chains for a single variable can be N^2, and thus the worst-case complexity of an algorithm that uses def-use chains is $O(N^2 \times V)$. In the SSA graph, however, only one definition reaches each use, and only N ϕ-functions can be inserted for each variable. This means that the worst-case complexity of

a constant-propagation algorithm that uses the SSA graph is only $O(N \times V)$.

6.5 Historical Notes

The first global constant propagation algorithm was presented by Kildall [Kil73] and also appears in [ASU86]. Several variations of this algorithm have also been published, and a generalization was published by Kam and Ullman [KU77]. Kildall's algorithm is roughly equivalent to the algorithm in Sec. 4.11.

We have described the problem in somewhat different terms from Kildall. In Kildall's lattice, each element corresponds to the state of all variables in the program. In our lattice, each element corresponds to the state of a single variable. By using our representation it is possible to lower the time bound to $O(E \times V)$ changes to variable values, which is the best one can hope to get out of an approach like this. These optimizations are not obvious in the context presented by Kildall. We have initialized the worklist with only the entry node, while Kildall started with all of the nodes on the worklist.

CC, which is based on Wegbreit's Algorithm 3.1 [Weg75], may be impractically slow and was ignored for a long time. Many workers in code optimization had tried to derive practical sparse algorithms that achieved CC's results. However, they started from the sparse representation then prevailing, namely def-use chains without SSA form. Def-use chains without SSA form do not determine the best information, as described in Sec. 6.4.

The maximal fixed point is the fixed point with the largest lattice elements at the nodes, and has been used as a minimal acceptable criterion of the quality of flow analysis (see Kam and Ullman [KU77] and Graham and Wegman [GW76]).

Several attempts have been made to formulate sparse versions of Kildall's algorithm. At least two pessimistic versions have been published and implemented, including an algorithm described by Kennedy in Chapter 6 of [CS70], an algorithm in Chapter 4 of [Ott78] and an algorithm by Kennedy in Chapter 1 of [MJ81]. Two optimistic versions have also been published, the first by Reif and Lewis [RL77, RL86] and the second by Ferrante and Ottenstein [FO83].

A precursor of SparseCC that did not use SSA form was presented by Wegman and Zadeck in [WZ85].

Acknowledgments

We would like to thank our colleagues Trina Avery, Jeff Barth, Larry Carter, Keith Cooper, Bill Harrison, Julian Padget, John Reif, Randy Scarborough, and G. A. Venkatesh for all the help they have given us.

Bibliography

[AC72] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, chapter 1, pages 1–30. Prentice Hall, 1972.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[CS70] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York U., April 1970.

[Ell86] J. Ellis. *Bulldog: a Compiler for VLIW Architectures*. MIT Press, 1986.

[FO83] J. Ferrante and K. J. Ottenstein. A program form based on data dependency in predicate regions. *Conf. Rec. Tenth ACM Symp. on Principles of Programming Languages*, pages 217–231, January 1983.

[FP82] M. Furtney and T. W. Pratt. Kernel-control tailoring of sequential programs for parallel execution. *Proc. 1982 IEEE Internat. Conf. on Parallel Processing*, pages 245–247, August 1982.

[GW76] S. L. Graham and M. N. Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, January 1976.

[JM76] N. D. Jones and S. S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. *Conf. Rec. Third ACM Symp. on Principles of Programming Languages*, pages 77–94, January 1976.

[Kil73] G. A. Kildall. A unified approach to global program optimization. *Conf. Rec. First ACM Symp. on Principles of Programming Languages*, pages 194–206, October 1973.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[MJ81] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis.* Prentice-Hall, 1981.

[Ott78] K. J. Ottenstein. Data-flow graphs as an intermediate form. Technical report, Dept. of Computer Science, Purdue U., August 1978.

[Pra78] T. W. Pratt. Program analysis and optimization through kernel-control decomposition. *Acta Informatica*, 9:195–216, 1978.

[RL77] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. *Conf. Rec. Fourth ACM Symp. on Principles of Programming Languages*, pages 104–118, January 1977.

[RL86] J. H. Reif and H. R. Lewis. Efficent symbolic analysis of programs. *J. Computer and System Sciences*, 32(3):280–313, June 1986.

[Weg75] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. on Software Engineering*, SE-1(3):270–285, September 1975.

[WZ85] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, pages 291–299, January 1985.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.